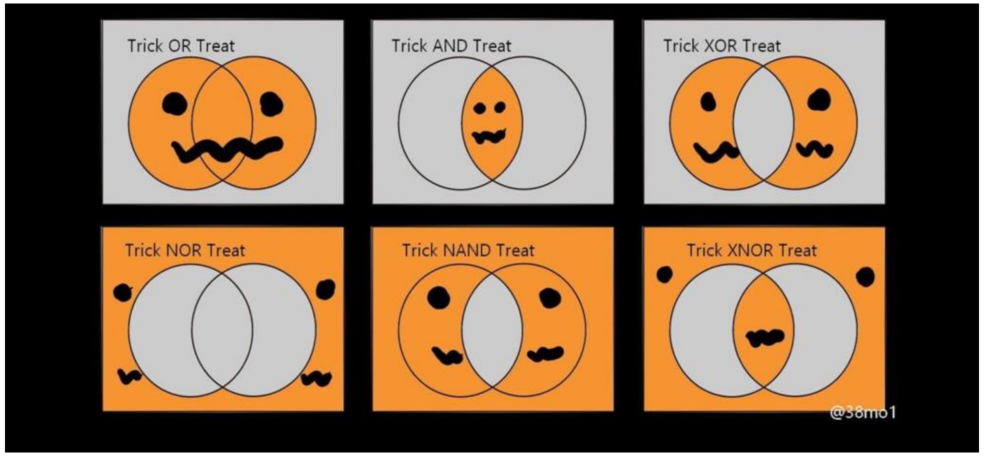
EECS 370

Cache Introduction



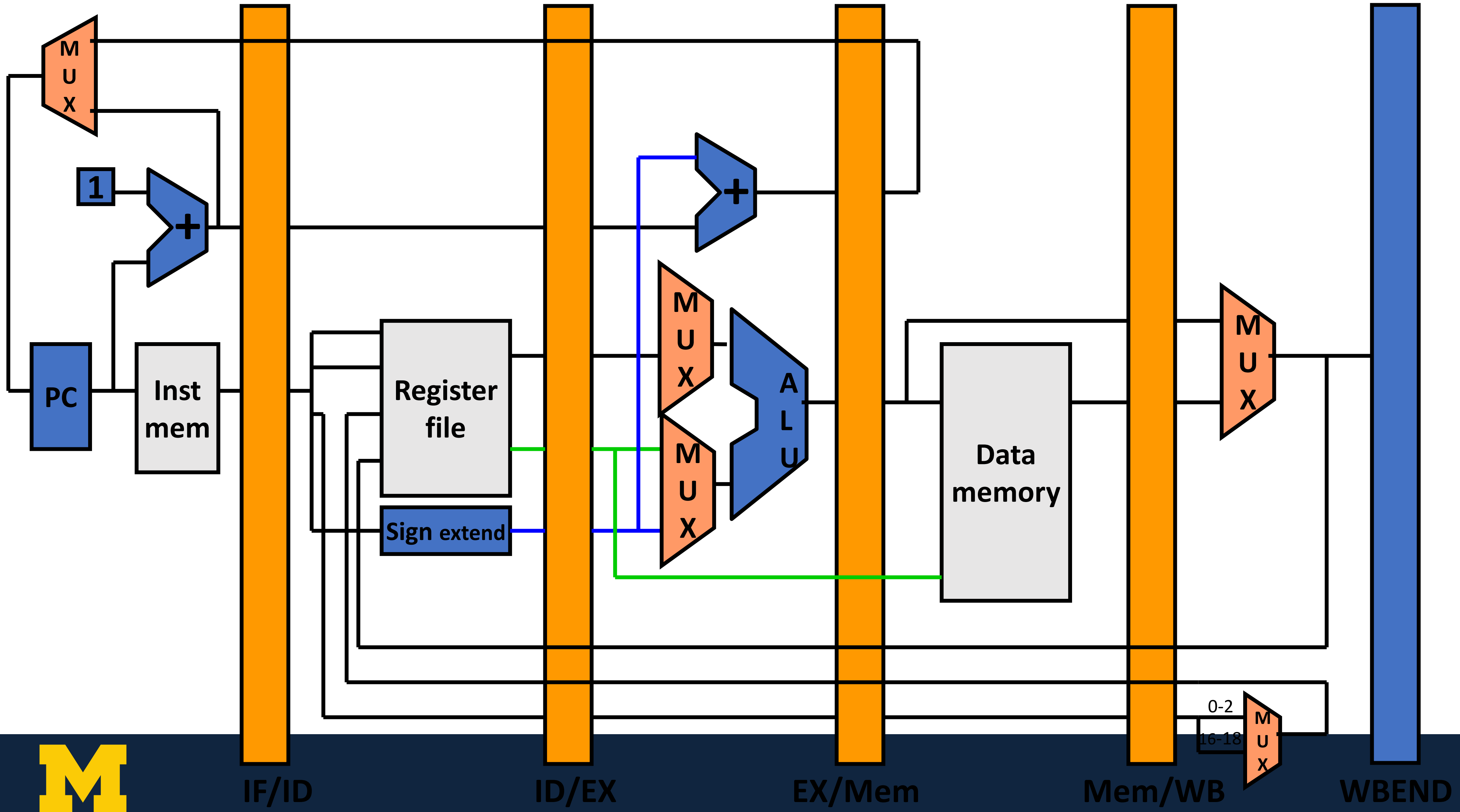
Announcements

- My office hours on Wed 11/8 are cancelled
- Lab 8 due Wed
  - Lab 9 meets Fr/M
- P3 Checkpoint due Thurs
  - Full P3 due Thu 11/9
- Homework 3 due Mon 11/6

Project 3 Pipeline

- Design is largely the same, except no "internal forwarding" through the register file
  - I.e. we need to explicitly forward from a new pipeline register WBEND if source instruction is in WB while dependent is in Decode

Project 3 Pipeline



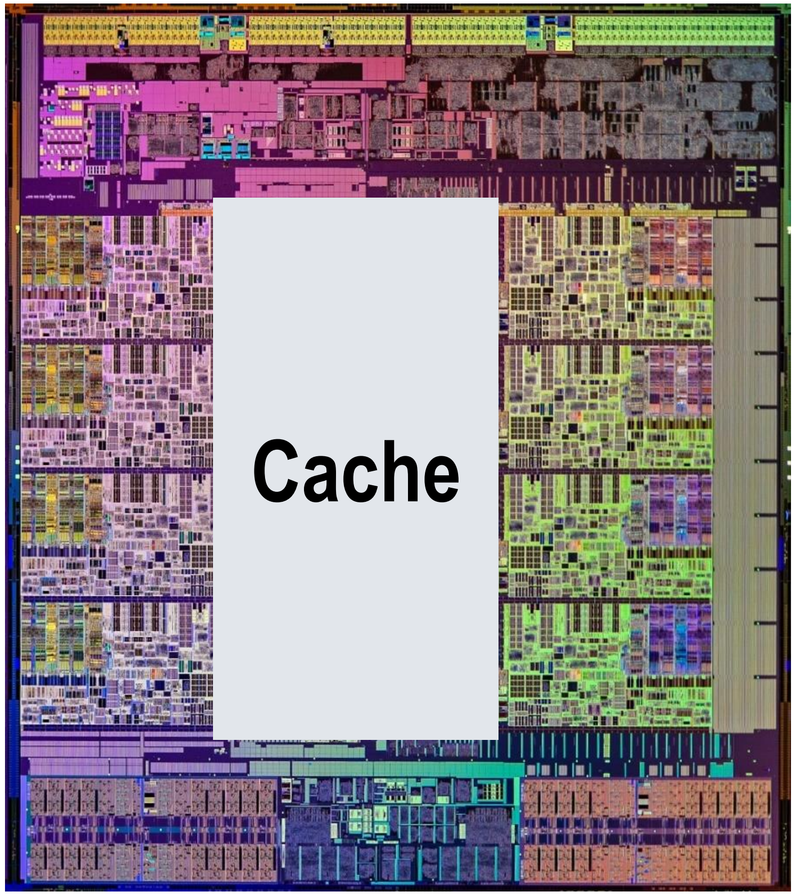
Agenda

- **Memory Types**
- Memory Hierarchy and Cache Principles
- Cache example
- How to improve cache

EECS 370 Overview

- Part I: Software
- Part II: Processor Design
- Part III: Memory Design
  - Starting with: caches

- If we judge a component's value by how much space it takes up on a chip (not a terrible heuristic), caches are **very** valuable



Cache Aware vs Non-Aware Code

```c
#include<stdio.h>
#include<stdlib.h>

#define N 20000
int arrayInt[N][N];

int main(int argc, char **argv)
{
  int i, j;
  int count = 0;

  for(i=0; i< N; i++)
    for(j = 0; j < N; j++ )
    {
        count++;
        arrayInt[i][j] = 10;
    }

    printf("Count :%d\n", count);
}
```

```c
#include<stdio.h>
#include<stdlib.h>

#define N 20000
int arrayInt[N][N];

int main(int argc, char **argv)
{
  int i, j;
  int count = 0;

  for(i=0; i< N; i++)
    for(j = 0; j < N; j++ )
    {
        count++;
        arrayInt[j][i] = 10;
    }

    printf("Count :%d\n", count);
}
```

Memory

- So far, we have discussed two structures that hold data:
  - Register file (little array of words)
  - Memory (bigger array of words)

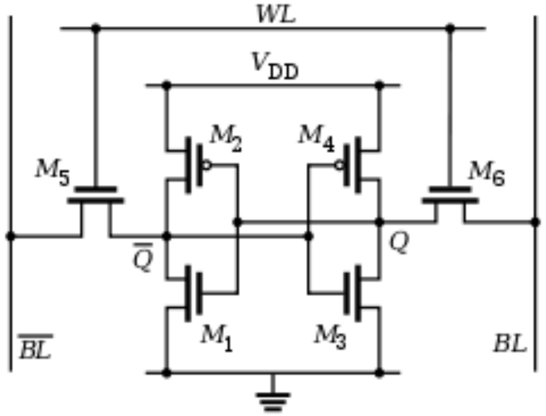
- How do we build this?
  - We need a lot of memory: 2^{18} for LC2K, a lot more for ARM
  - Bunch of flip-flops? Not practical – too many transistors, would be huge and power hungry
  - Other, clever ways of storing bits

when the bits is small, flip-flop is fast

Option 1: SRAM

Static Random access memory.

- Each bit is made of 6 transistors
- Volatile: need constant power to keep data
- Fast: ~1 ns read time
- Still rather large
  - Can only put ~MBs on chip
  - Impractical to scale up to GBs needed for memory



Option 2: DRAM

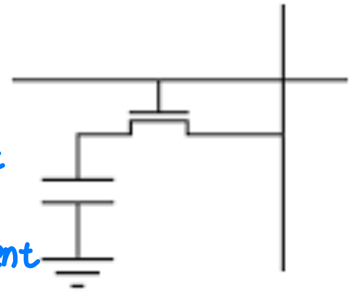
Dynamic

因为 capacitor 里面存的电压
会不断下降，所以要不断补充

- Each bit is made of a transistor and capacitor
- Volatile: need constant power to keep data
- Slower: ~50 ns access time

同时也是慢的原因.

  - Must stall for dozens of cycles on each memory load
- Less expensive for SRAM
  - We can put up to ~16-64 GB in current machines
  - Good for LC2K or 32-bit systems
  - But not for 64-bit systems



have voltage different (charged) : 1

Don't have voltage different (uncharged) : 0

Option 3: Disks

- Hard-drives store bits as magnetic charges on spinning disks
  - Non-volatile – holds information even when no power is supplied
  - Obnoxiously slow compared to digital logic: 3,000,000 ns access time
- Recently, solid-state drives have replaced spinning disks with logic gates replacing mechanical systems
  - Also non-volatile
  - Much better speeds (3,000,000 ns), but still to slow to keep up with processor
- Cheap!
  - SSDs cost $0.0001 per megabyte
  - Scale up to terabytes – practical for modern computing

Agenda

- Memory Types
- **Memory Hierarchy and Cache Principles**
- Cache example
- How to improve cache

Memory Goals

- Fast: Ideally run at processor clock speed
  - 1 ns access
- Cheap: Ideally free
  - Not more expensive than rest of system

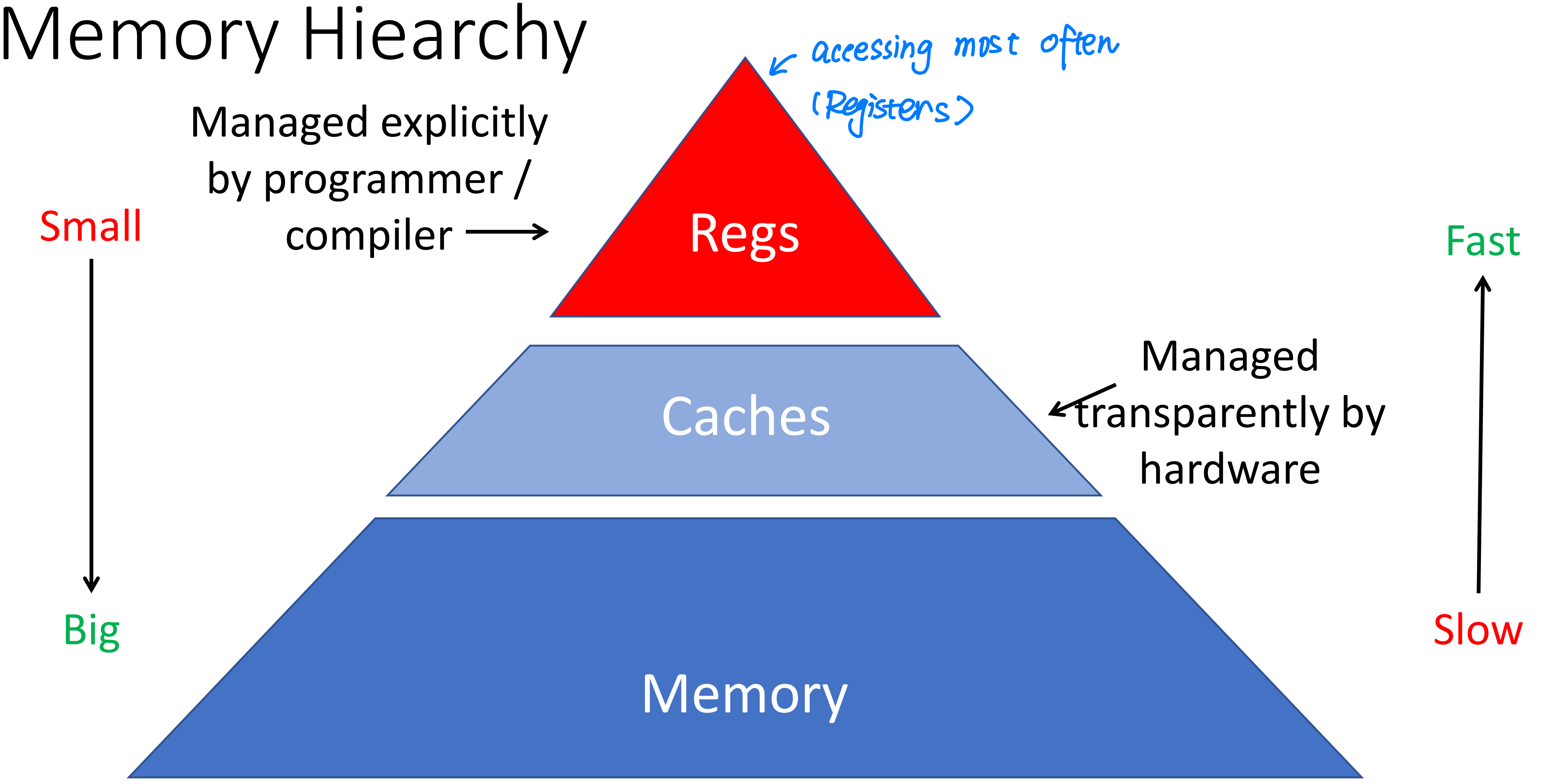
- DRAM, hard-drives and SSDs are too slow
- SRAM is too expensive

- How to get best properties of multiple memory technologies?

Memory Hierarchy

- Key observation: we only need to access a small amount of data at a time
- Let's use a small array of SRAM to hold data we need now
  - Call this the **cache**
  - Able to keep up with processor
  - Small (~Kilobytes), so it should be relatively cheap
- Use a large amount of DRAM for **main memory**
  - Can scale up to ~Gigabytes in size
- Everything else, store on disk
  - **Virtual Memory**
- Won't end up building 2^{64} of anything
  - Won't be needed in typical programs
  - Virtual memory (discussed in a couple weeks) will make memory look larger than it is

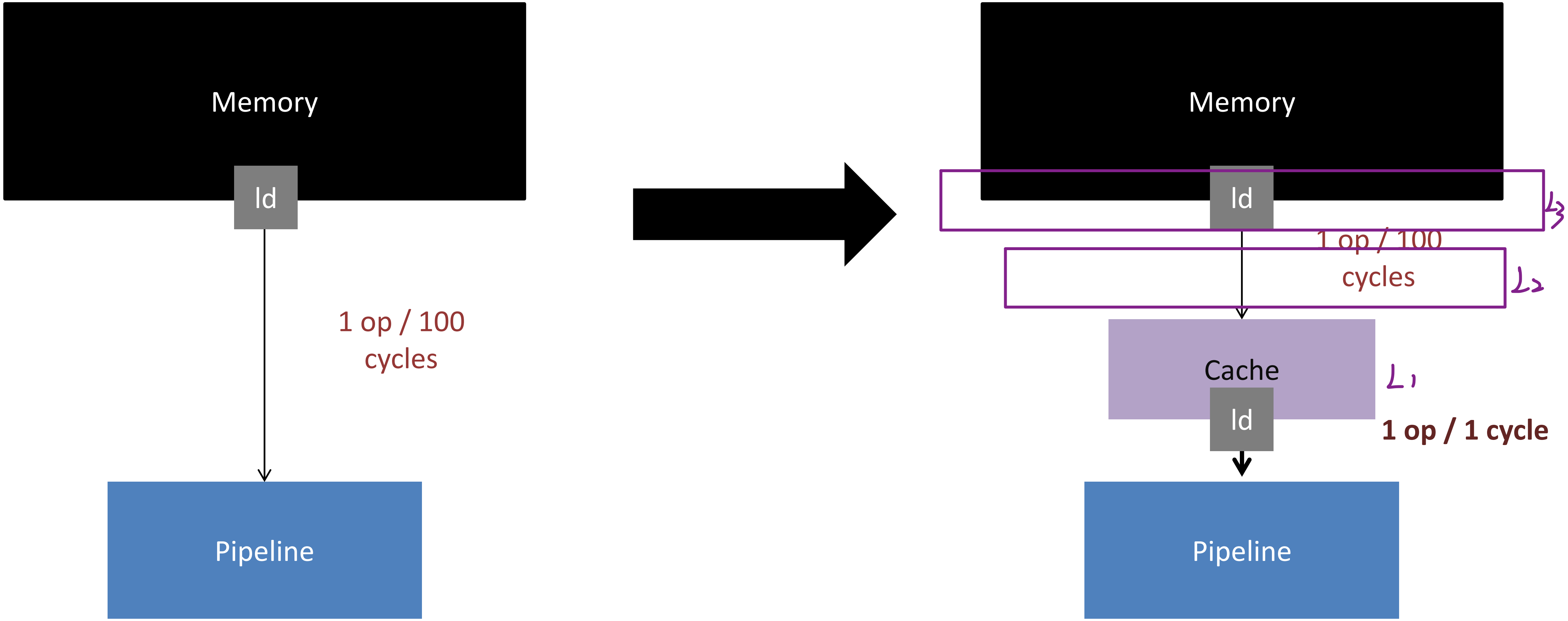
Memory Hiearchy


accessing most often
(Registers)
Managed explicitly by programmer / compiler
Small
Big
Regs
Caches
Memory
Managed transparently by hardware
Fast
Slow

Cache Analogy

- Studying in the library
  - **Option 1: Every time you grab another book, return current book to shelf and get a new book from shelf**
    - Latency = 5 minutes
  - **Option 2: Keep 10 commonly-used books on shelf above desk**
    - Latency = 1 minute
  - **Option 3: Keep 3 books open on different locations on desk**
    - Latency = 10 seconds

Caches - Overview

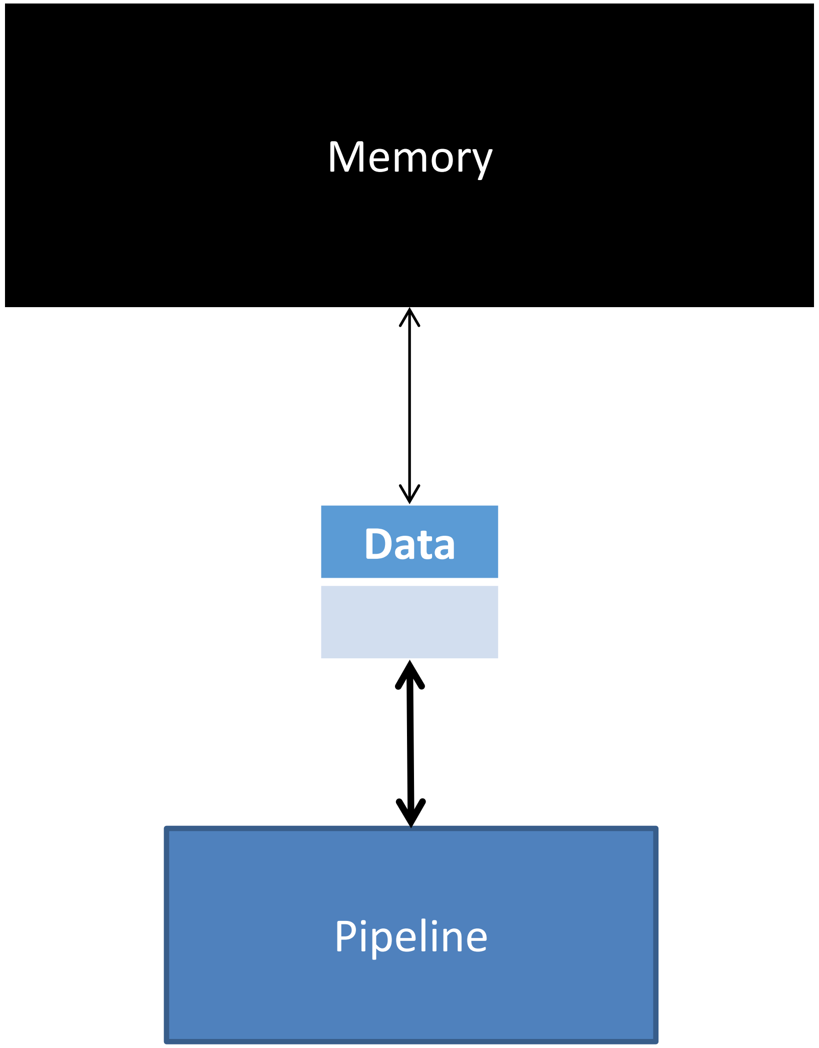


Function of the Cache

- The cache holds the data we think is **most likely** to be referenced
  - The more often the data we want is in the fast cache, the lower our **average memory access latency** is
  - How do we decide what the most likely accessed memory locations are?

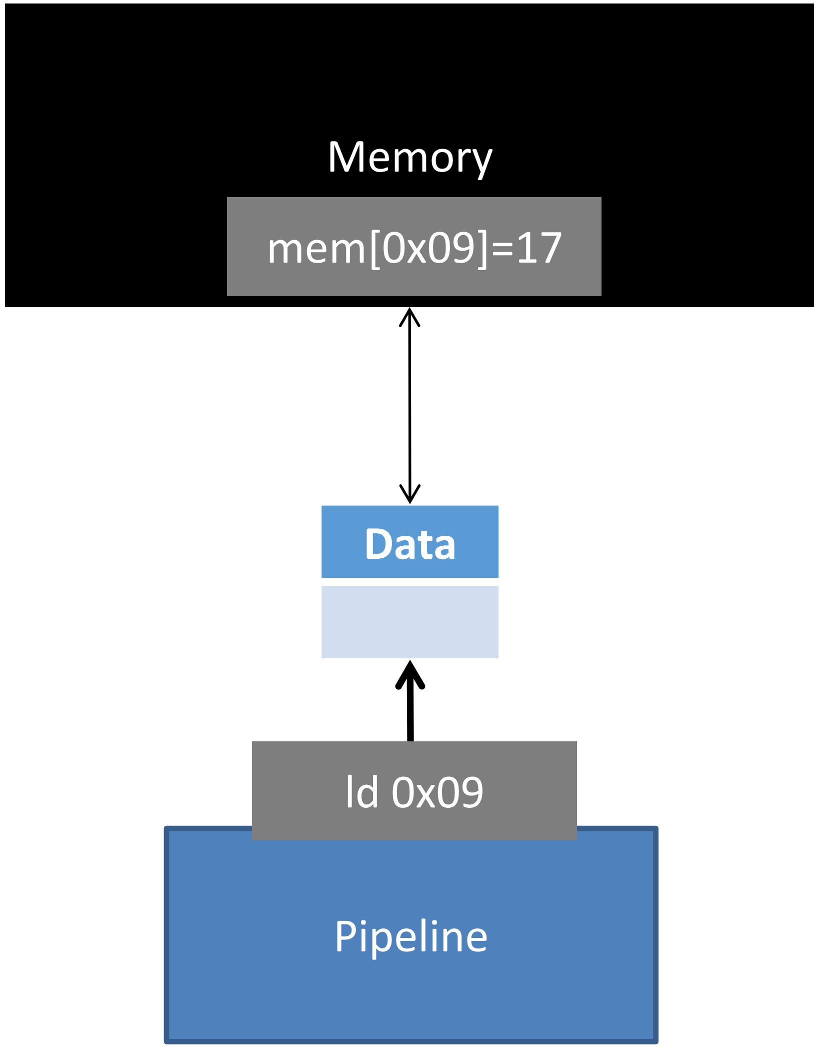
The simplest cache

- Only worry about load instructions for now
- Word-addressable address space
- Consists of a single, word-size storage location to remember last loaded value



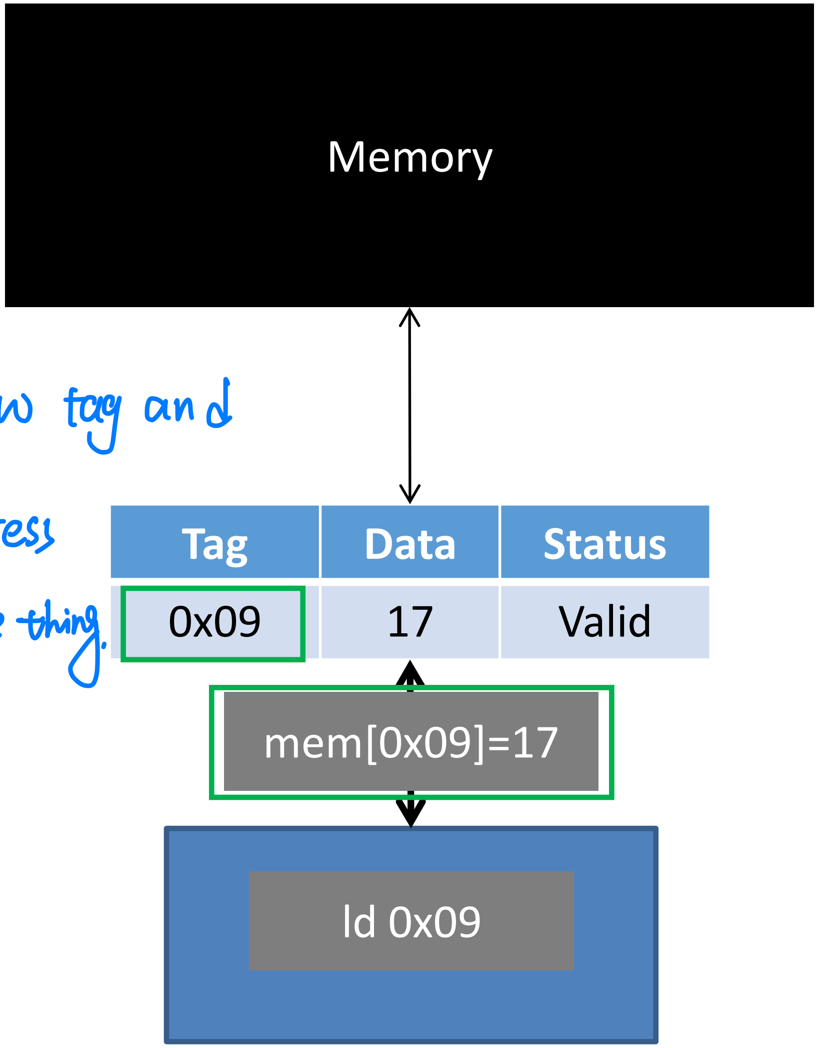
The simplest cache

- Whenever memory returns data, store it in the cache
- We'll also need to know what address this data corresponds to
  - Store that as "**tag**"
- Also include a "**valid**" status bit



The simplest cache

- Next memory access, first check if the tag matches address
    - Yes? Return cache data
    - No? Go to memory as before



Tag	Data	Status
0x09	17	Valid

right now tag and memory address are the same thing.

Agenda

- Memory Types
- Memory Hierarchy and Cache Principles
- **Cache example**
- How to improve cache

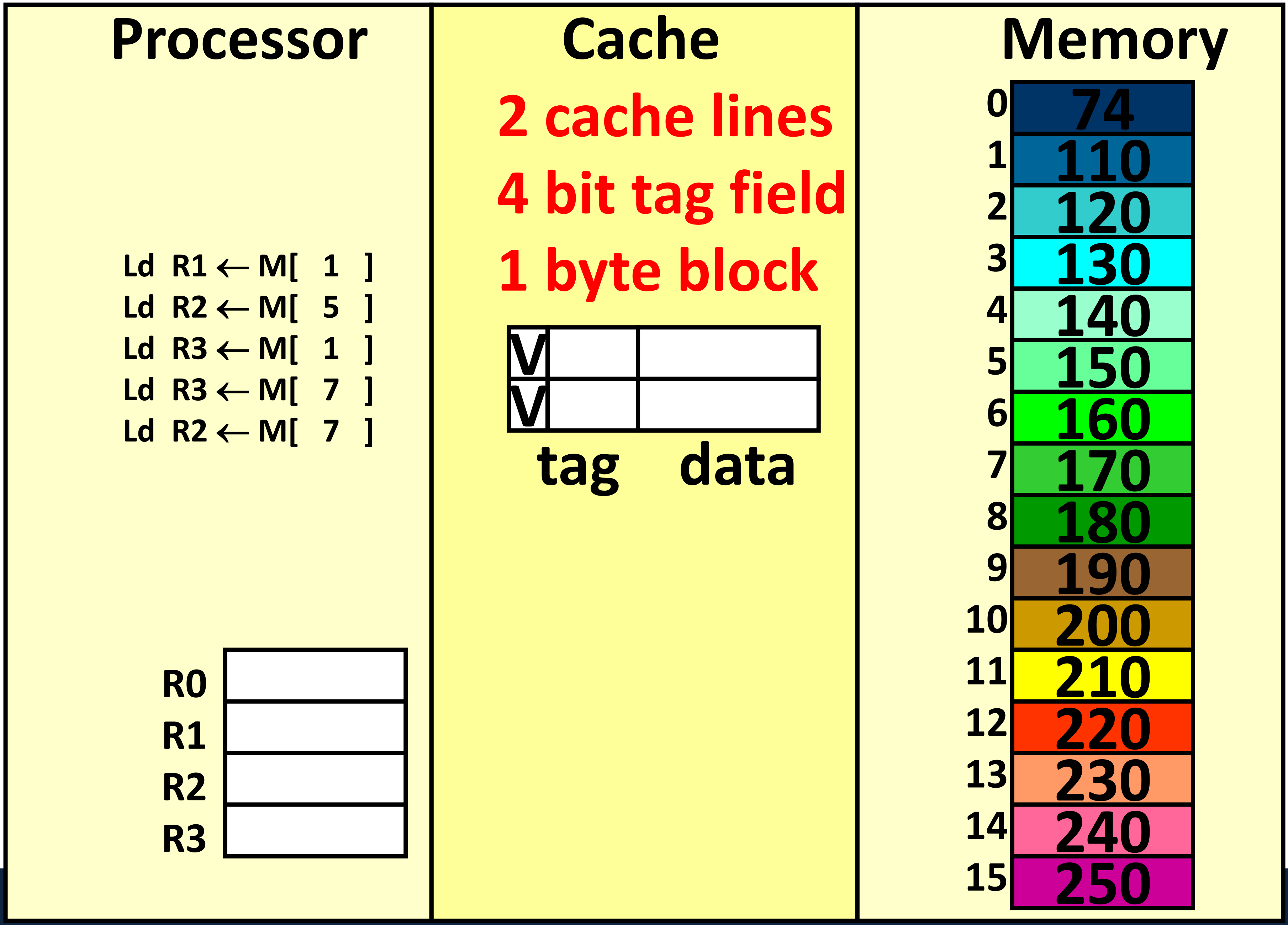
Scaling Up

- Of course, we don't just access one memory location
- What if we access many memory locations, and cache isn't large enough to hold all of them?
- How do we choose what to keep in the cache?
  - How does hardware predict what's most likely to be needed soon?
- Answer: **locality**
  - Temporal locality
  - Spatial locality

Temporal Locality

- Temporal locality: if a given memory location is referenced now, it will probably be used again in the near future
  - Why? Take a look at code you've written. You tend to use a variable multiple times
  - Corollary: if you haven't used a variable in a while, you probably won't need it very soon either
- Hardware should take advantage of this by:
  - Placing items we just accessed in the cache
  - When we need to evict something, evict whatever data was **least recently used (LRU)**

A Very Simple Memory System



Processor

```
Ld  R1 ← M[  1  ]
Ld  R2 ← M[  5  ]
Ld  R3 ← M[  1  ]
Ld  R3 ← M[  7  ]
Ld  R2 ← M[  7  ]
```

R0	
R1	
R2	
R3	

Cache

2 cache lines

4 bit tag field

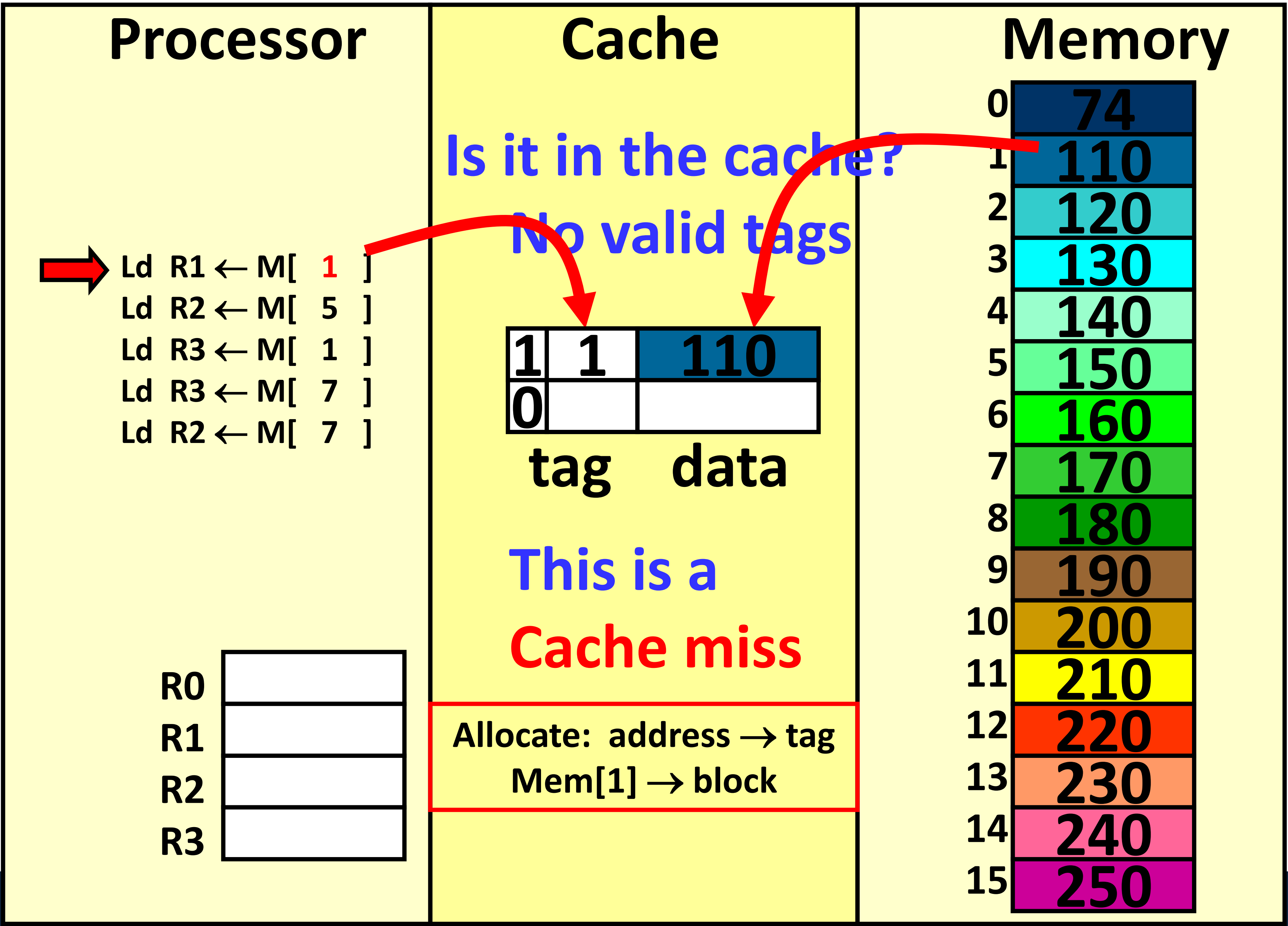
1 byte block

	tag	data
V		
V		

Memory

0	74
1	110
2	120
3	130
4	140
5	150
6	160
7	170
8	180
9	190
10	200
11	210
12	220
13	230
14	240
15	250

A Very Simple Memory System



Processor

Ld R1 ← M[1]
Ld R2 ← M[5]
Ld R3 ← M[1]
Ld R3 ← M[7]
Ld R2 ← M[7]

R0	
R1	
R2	
R3	

Cache

Is it in the cache?

No valid tags

	tag	data
1	1	110
0		

This is a

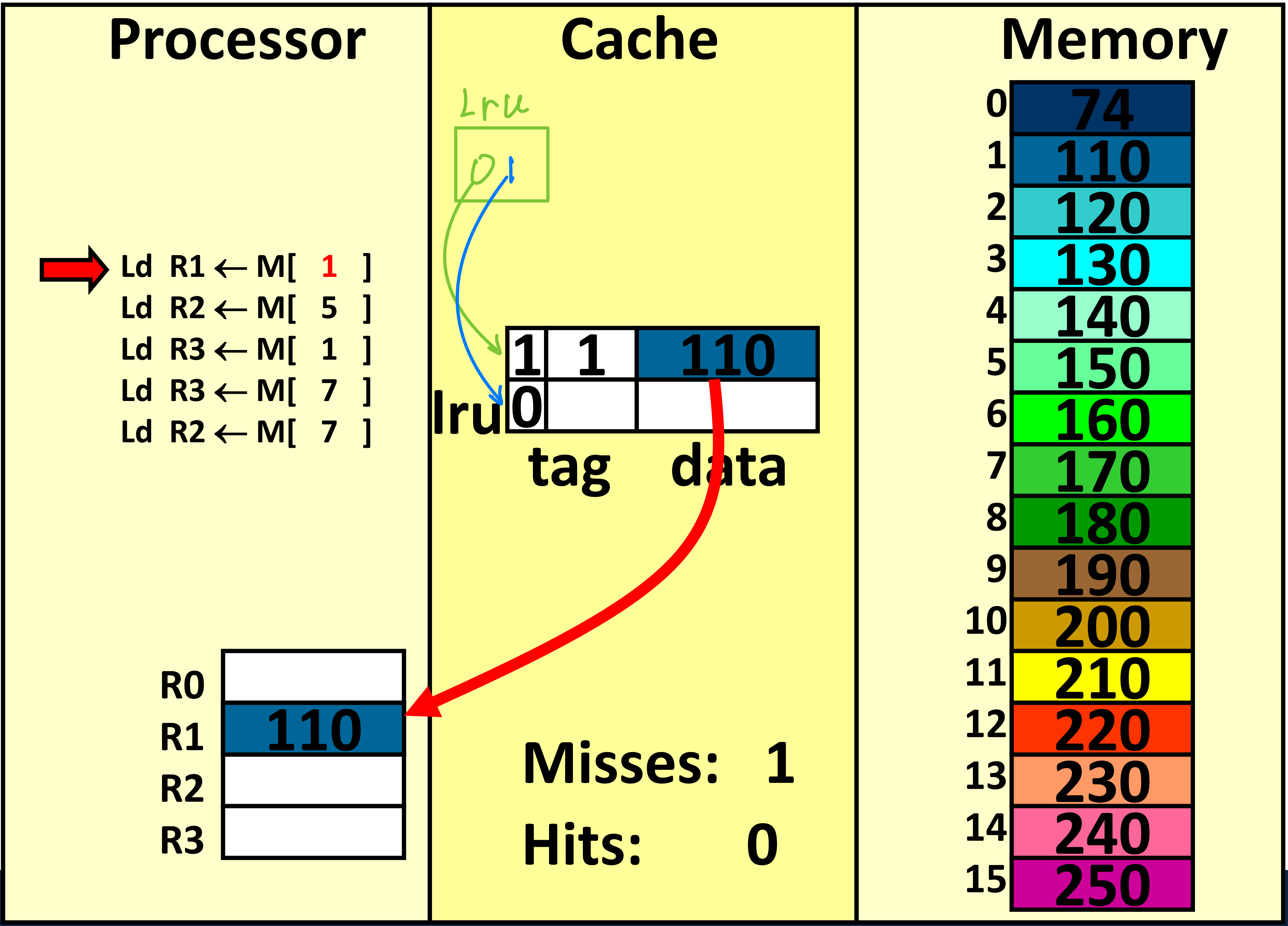
Cache miss

Allocate: address → tag
Mem[1] → block

Memory

0	74
1	110
2	120
3	130
4	140
5	150
6	160
7	170
8	180
9	190
10	200
11	210
12	220
13	230
14	240
15	250

A Very Simple Memory System



Processor

```
Ld  R1 ← M[  1  ]
Ld  R2 ← M[  5  ]
Ld  R3 ← M[  1  ]
Ld  R3 ← M[  7  ]
Ld  R2 ← M[  7  ]
```

R0	
R1	110
R2	
R3	

Cache

Lru

0 1

lru	tag	data
1	1	110
0		

Misses: 1

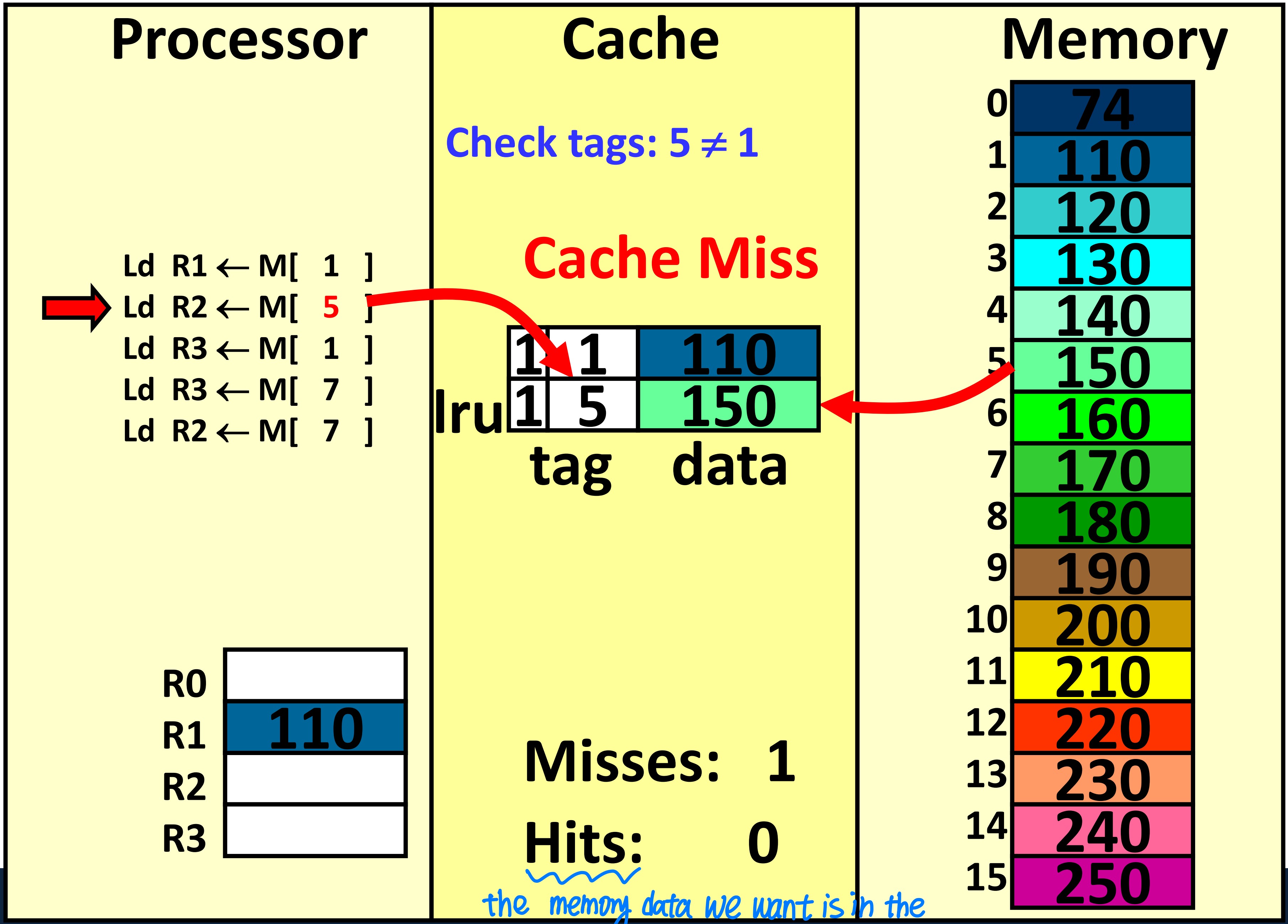
Hits: 0

Memory

0	74
1	110
2	120
3	130
4	140
5	150
6	160
7	170
8	180
9	190
10	200
11	210
12	220
13	230
14	240
15	250

A Very Simple Memory System

Tag comparison uses hardware called "content-addressable memory (CAM)"



Processor

Ld R1 ← M[1]
Ld R2 ← M[5]
Ld R3 ← M[1]
Ld R3 ← M[7]
Ld R2 ← M[7]

R0	
R1	110
R2	
R3	

Cache

Check tags: 5 ≠ 1

Cache Miss

lru	tag	data
1	1	110
1	5	150

Misses: 1

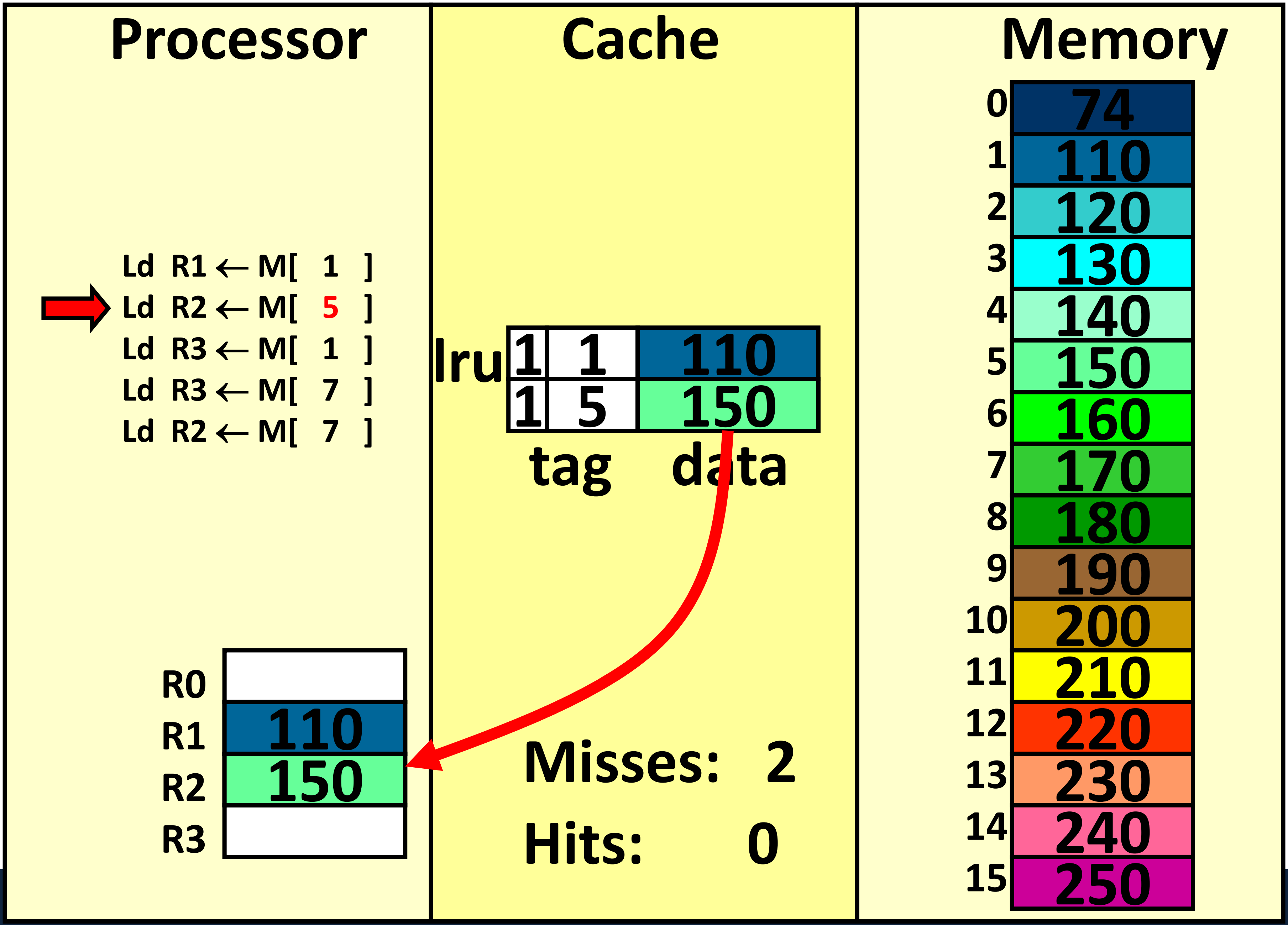
Hits: 0

Memory

0	74
1	110
2	120
3	130
4	140
5	150
6	160
7	170
8	180
9	190
10	200
11	210
12	220
13	230
14	240
15	250

the memory data we want is in the cache .

A Very Simple Memory System



Processor

Ld R1 ← M[1]
Ld R2 ← M[5]
Ld R3 ← M[1]
Ld R3 ← M[7]
Ld R2 ← M[7]

R0	
R1	110
R2	150
R3	

Cache

lru		tag	data
	1	1	110
	1	5	150

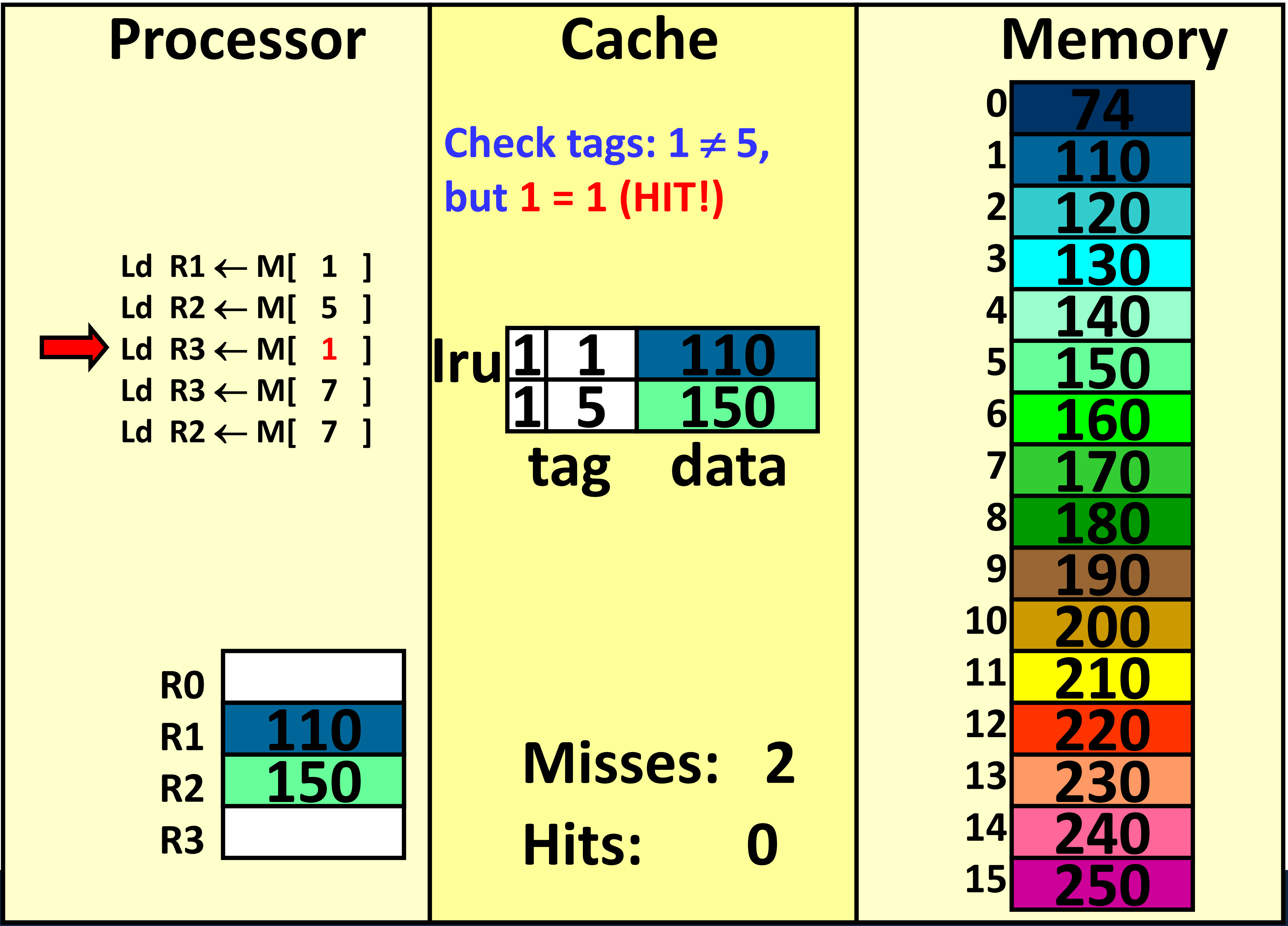
Misses: 2

Hits: 0

Memory

0	74
1	110
2	120
3	130
4	140
5	150
6	160
7	170
8	180
9	190
10	200
11	210
12	220
13	230
14	240
15	250

A Very Simple Memory System



Processor

```
    Ld  R1 ← M[  1  ]
    Ld  R2 ← M[  5  ]
➡   Ld  R3 ← M[  1  ]
    Ld  R3 ← M[  7  ]
    Ld  R2 ← M[  7  ]
```

R0	
R1	110
R2	150
R3	

Cache

Check tags: 1 ≠ 5, but 1 = 1 (HIT!)

lru	tag	data
1	1	110
1	5	150

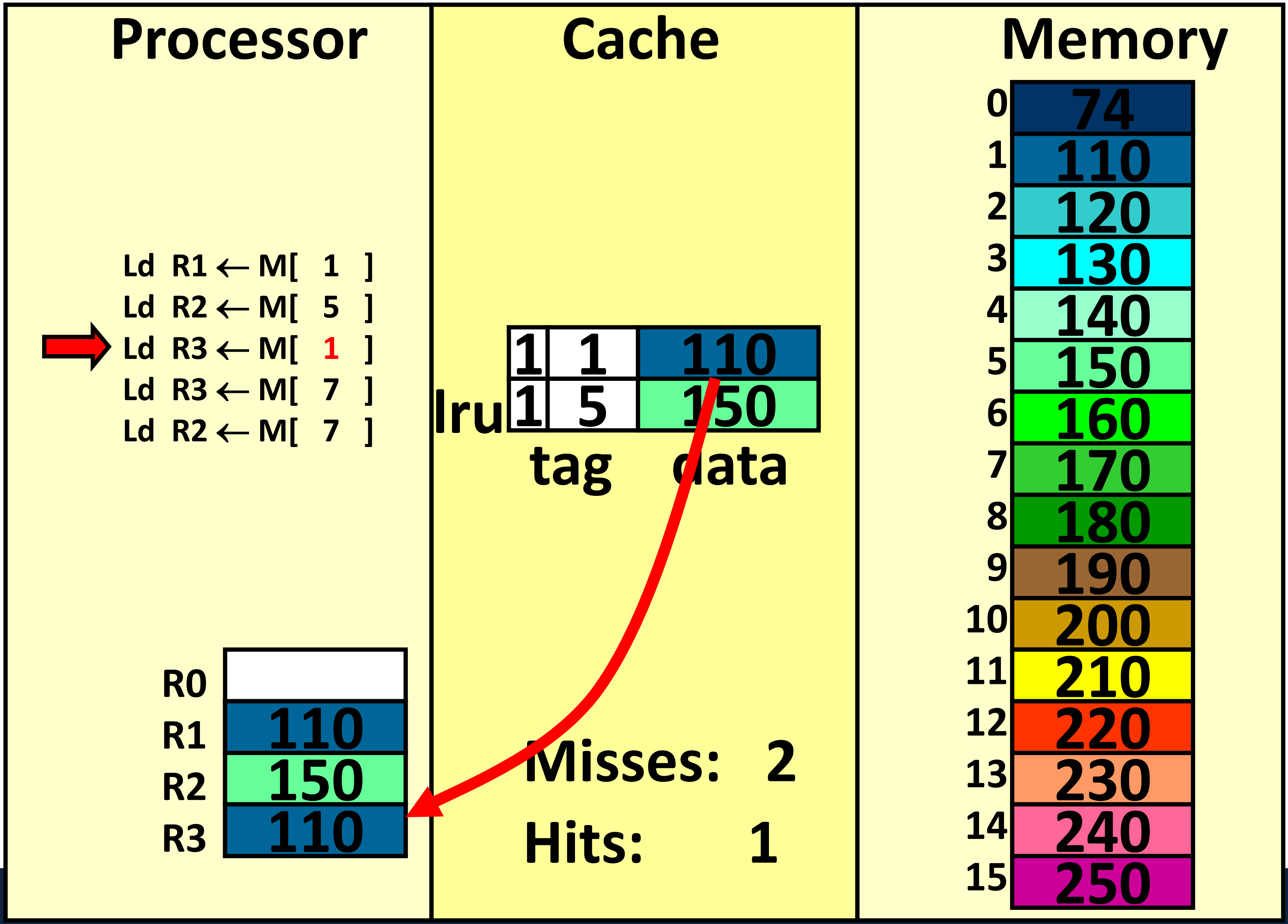
Misses: 2

Hits: 0

Memory

0	74
1	110
2	120
3	130
4	140
5	150
6	160
7	170
8	180
9	190
10	200
11	210
12	220
13	230
14	240
15	250

A Very Simple Memory System



Processor

```
Ld  R1 ← M[  1  ]
Ld  R2 ← M[  5  ]
Ld  R3 ← M[  1  ]
Ld  R3 ← M[  7  ]
Ld  R2 ← M[  7  ]
```

R0	
R1	110
R2	150
R3	110

Cache

		tag	data
	1	1	110
lru	1	5	150

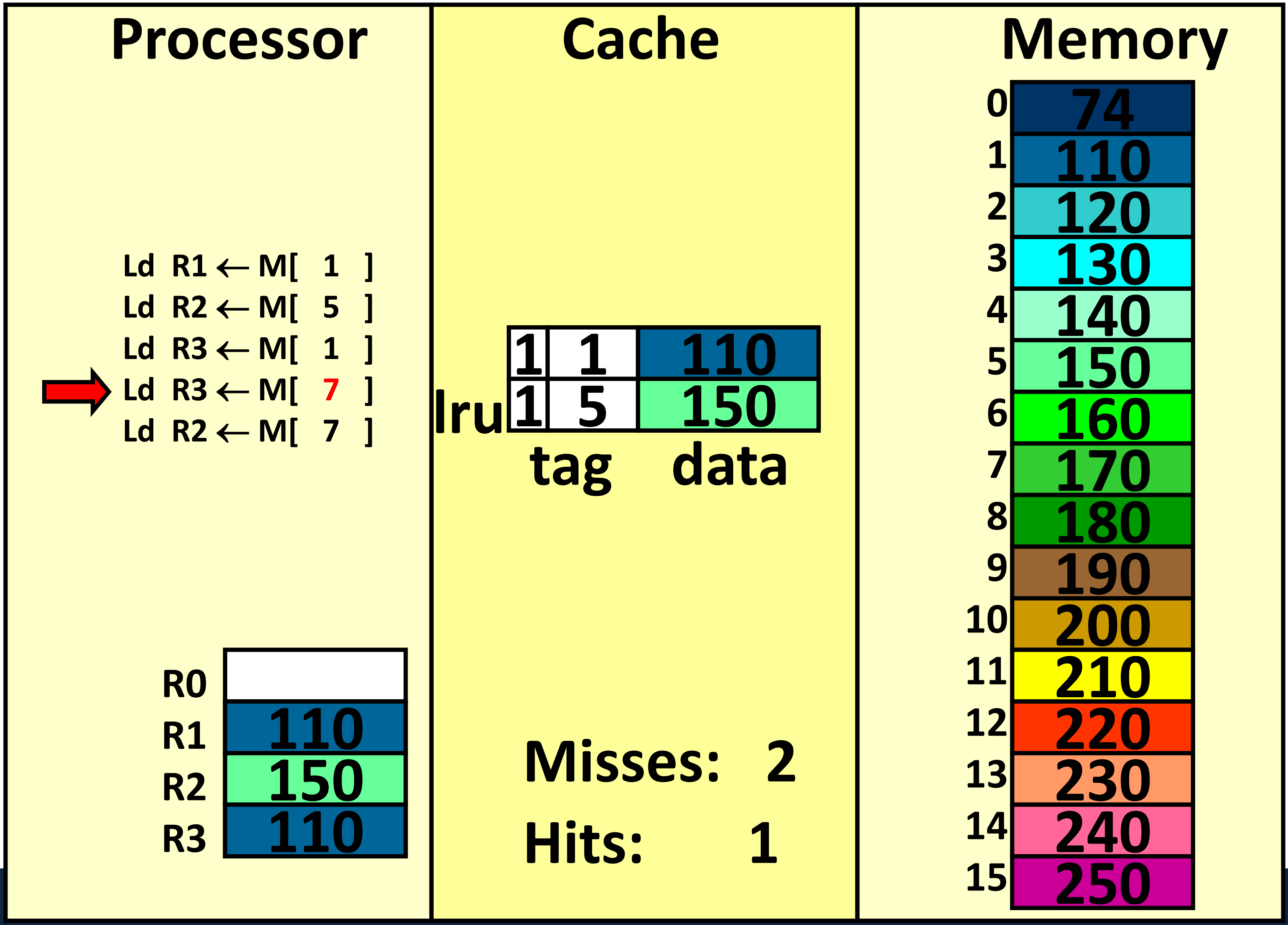
Misses: 2

Hits: 1

Memory

0	74
1	110
2	120
3	130
4	140
5	150
6	160
7	170
8	180
9	190
10	200
11	210
12	220
13	230
14	240
15	250

A Very Simple Memory System



Processor

```
Ld  R1 ← M[  1  ]
Ld  R2 ← M[  5  ]
Ld  R3 ← M[  1  ]
Ld  R3 ← M[  7  ]
Ld  R2 ← M[  7  ]
```

R0	
R1	110
R2	150
R3	110

Cache

	1	1	110
lru	1	5	150
		tag	data

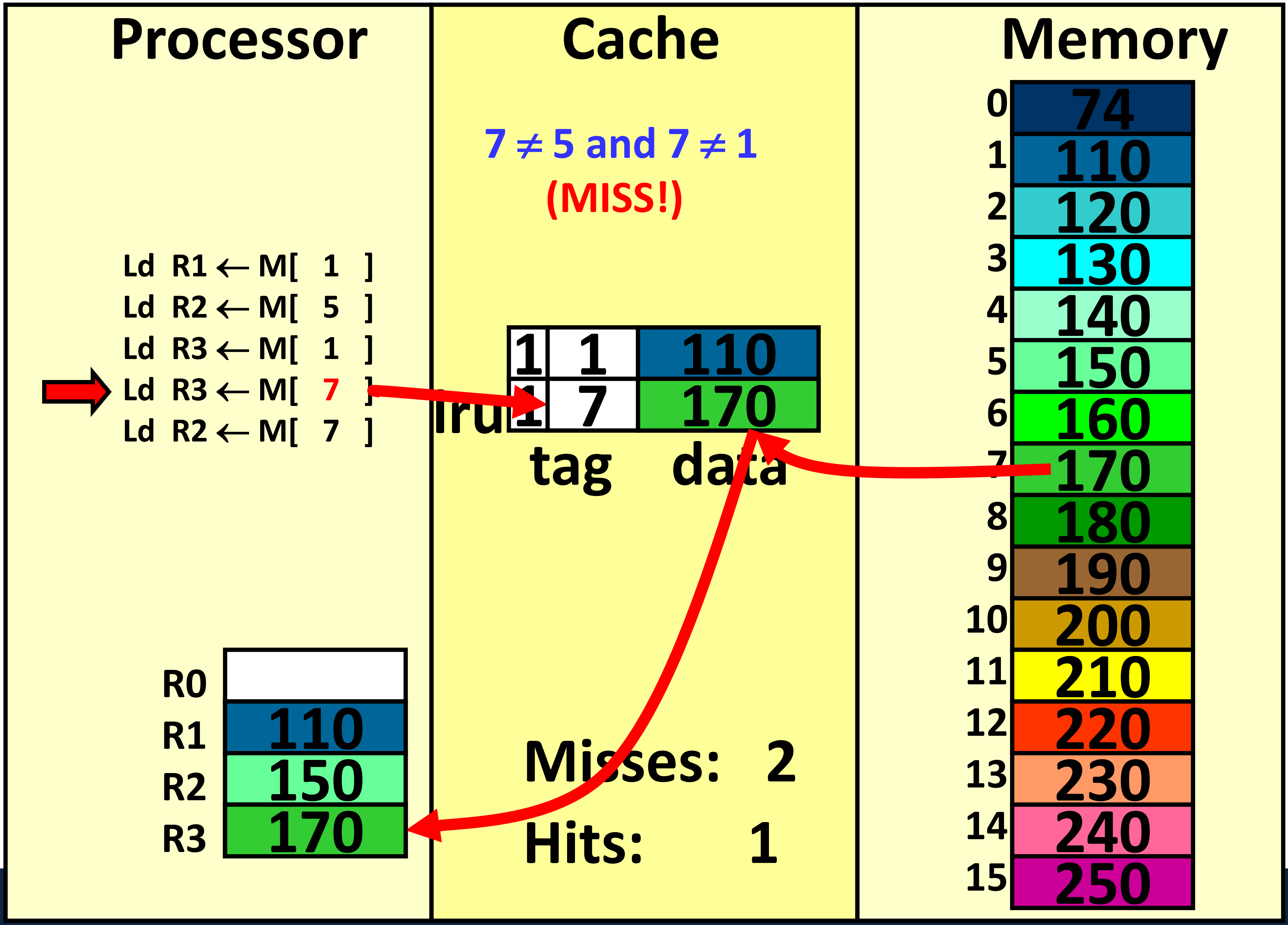
Misses: 2

Hits: 1

Memory

0	74
1	110
2	120
3	130
4	140
5	150
6	160
7	170
8	180
9	190
10	200
11	210
12	220
13	230
14	240
15	250

A Very Simple Memory System



Processor

```
Ld R1 ← M[ 1 ]
Ld R2 ← M[ 5 ]
Ld R3 ← M[ 1 ]
Ld R3 ← M[ 7 ]
Ld R2 ← M[ 7 ]
```

R0	
R1	110
R2	150
R3	170

Cache

7 ≠ 5 and 7 ≠ 1

(MISS!)

lru	tag	data
1	1	110
1	7	170

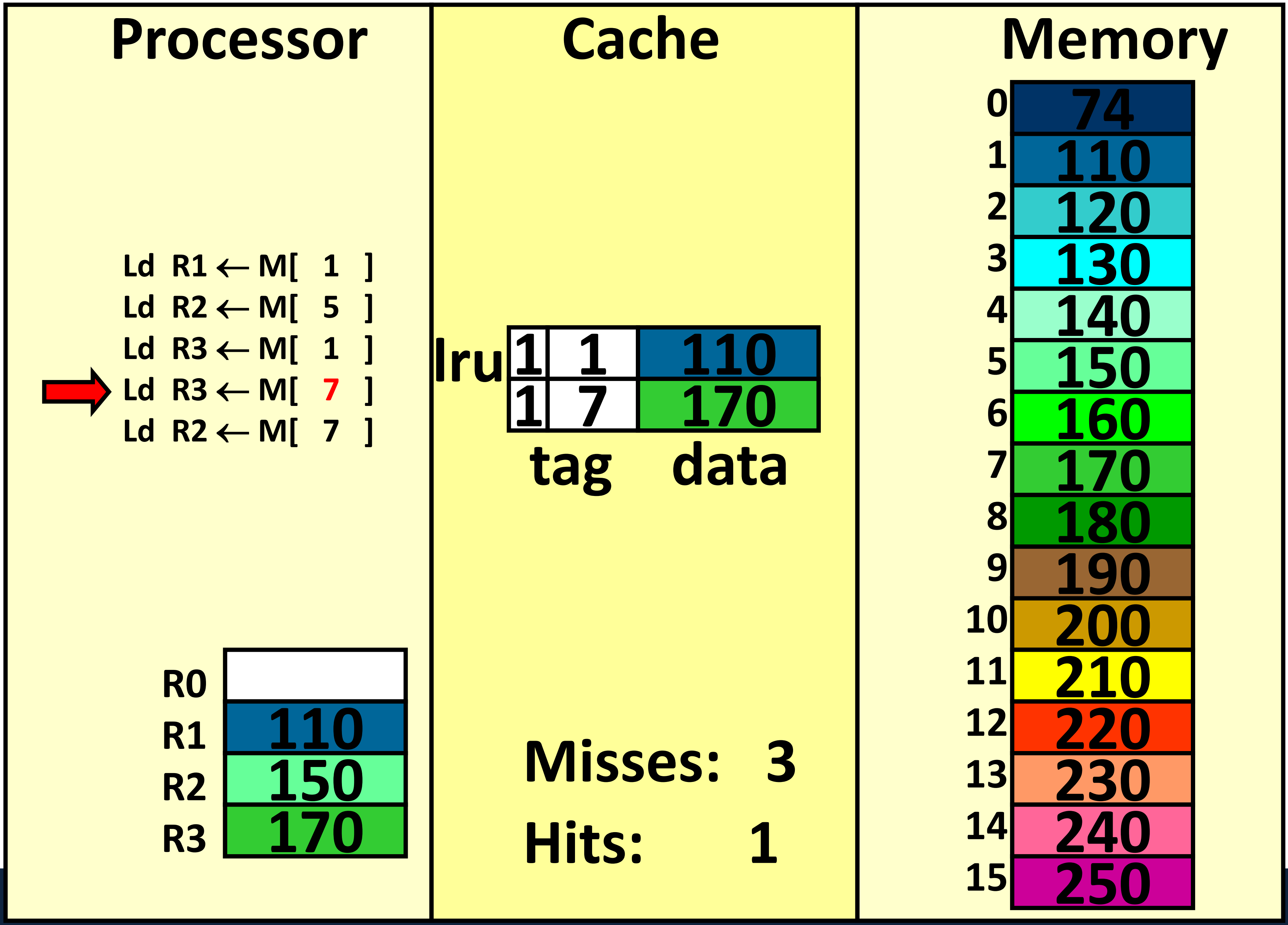
Misses: 2

Hits: 1

Memory

0	74
1	110
2	120
3	130
4	140
5	150
6	160
7	170
8	180
9	190
10	200
11	210
12	220
13	230
14	240
15	250

A Very Simple Memory System



Processor

```
    Ld  R1 ← M[  1  ]
    Ld  R2 ← M[  5  ]
    Ld  R3 ← M[  1  ]
==> Ld  R3 ← M[  7  ]
    Ld  R2 ← M[  7  ]
```

R0	
R1	110
R2	150
R3	170

Cache

lru	tag	data
1	1	110
1	7	170

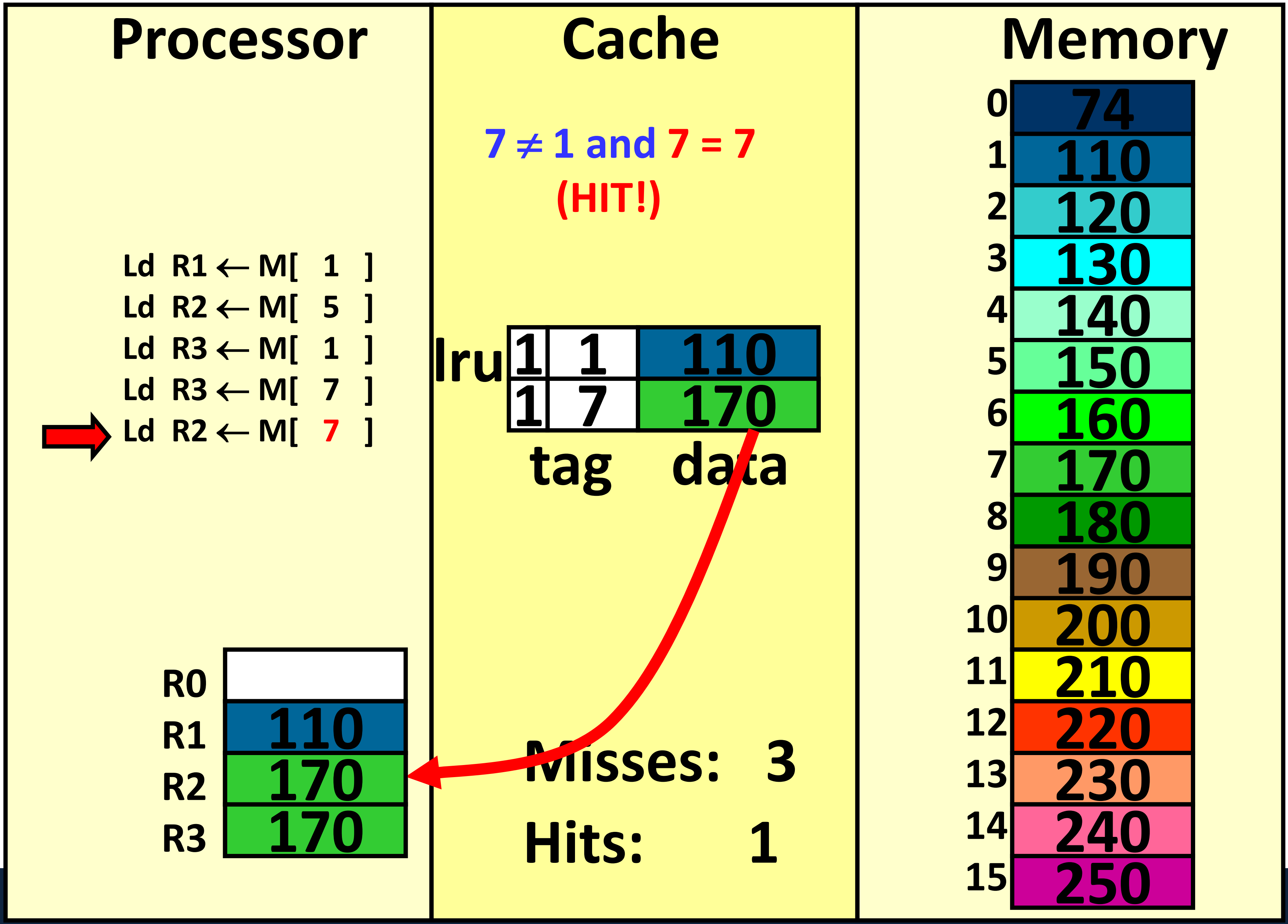
Misses: 3

Hits: 1

Memory

0	74
1	110
2	120
3	130
4	140
5	150
6	160
7	170
8	180
9	190
10	200
11	210
12	220
13	230
14	240
15	250

A Very Simple Memory System



Processor

```
Ld  R1 ← M[  1  ]
Ld  R2 ← M[  5  ]
Ld  R3 ← M[  1  ]
Ld  R3 ← M[  7  ]
Ld  R2 ← M[  7  ]
```

R0	
R1	110
R2	170
R3	170

Cache

7 ≠ 1 and 7 = 7

(HIT!)

lru	tag	data
1	1	110
1	7	170

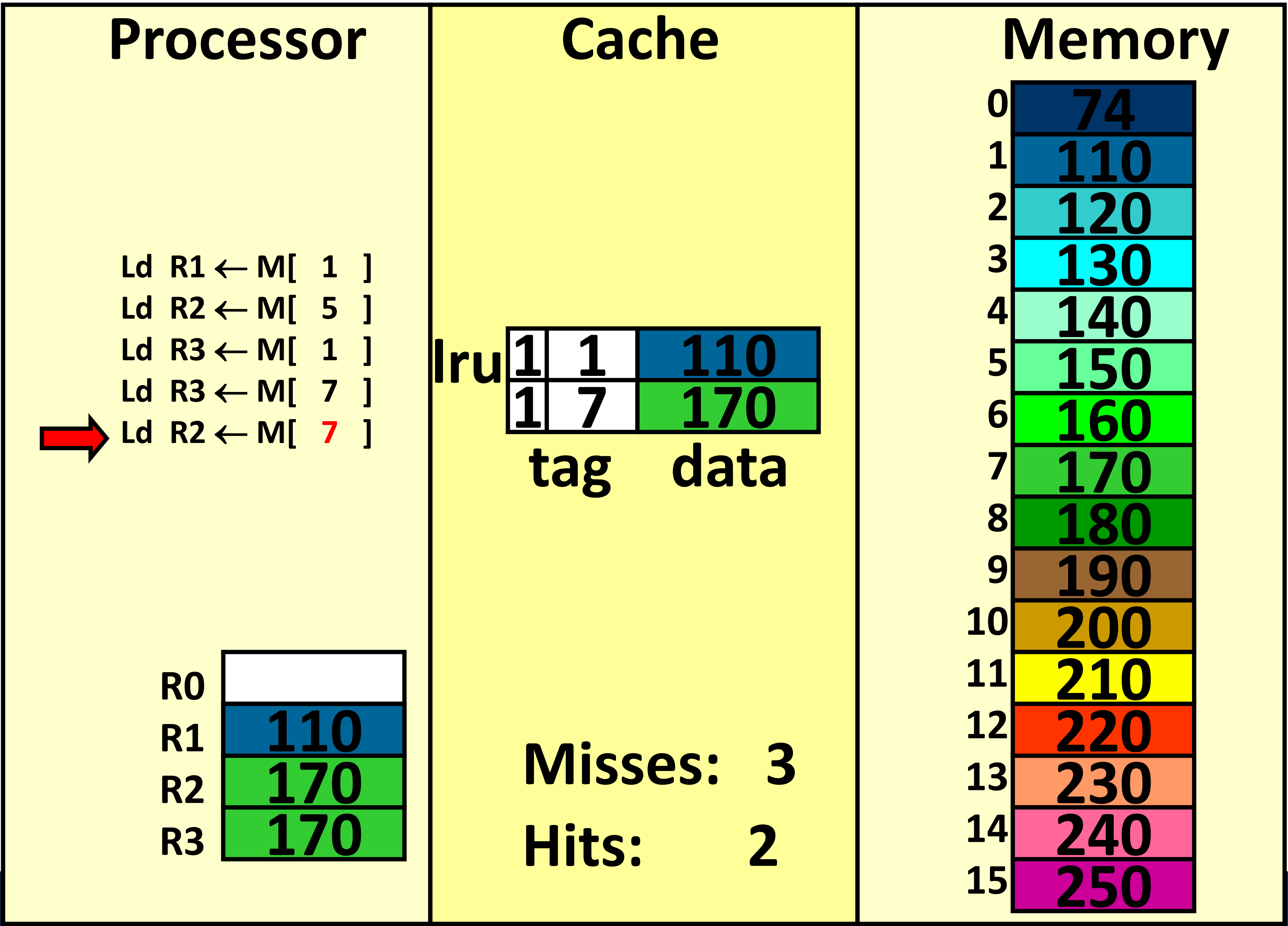
Misses: 3

Hits: 1

Memory

0	74
1	110
2	120
3	130
4	140
5	150
6	160
7	170
8	180
9	190
10	200
11	210
12	220
13	230
14	240
15	250

A Very Simple Memory System



Processor

```
Ld  R1 ← M[  1  ]
Ld  R2 ← M[  5  ]
Ld  R3 ← M[  1  ]
Ld  R3 ← M[  7  ]
Ld  R2 ← M[  7  ]
```

R0	
R1	110
R2	170
R3	170

Cache

lru	tag	data
1	1	110
1	7	170

Misses: 3

Hits: 2

Memory

0	74
1	110
2	120
3	130
4	140
5	150
6	160
7	170
8	180
9	190
10	200
11	210
12	220
13	230
14	240
15	250

Agenda

- Memory Types
- Memory Hierarchy and Cache Principles
- Cache example
- **How to improve cache**

Definitions

- Hit: when data for a memory access is found in the cache
- Miss: when data for a memory access is not found in the cache
- Hit/Miss rate: percentage of memory accesses that hit/miss in the cache

Poll: Average access time?

Example Problem

- Assume the following:
  - Cache has 1 cycle access time
  - If data is not found in cache, main memory is then accessed instead
  - Main memory has 100 cycle access time

- If we have a 90% hit rate in the cache, what is the average memory latency?

$0.9 \times 1 + 0.1 \times 101 = 11$

$1 \times 1 + 0.1 \times 100 = 11$

1 + 0.1 * (100) = 11

Calculating Average Access Latency

- Average Latency:
  - $cache\ latency + (memory\ latency \cdot miss\ rate)$
- Average latency for our example:
  - $1\ cycle + \left(15 \cdot \frac{3}{5}\right)$
  - $= 10\ cycles\ per\ reference$

- To improve latency, either:
  - Improve memory access latency, or
  - Improve cache access latency, or
  - Improve cache hit rate

Next time

- Making our caches bigger and better